FlushTime: Towards Mitigating Flush-based Cache Attacks via Collaborating Flush Instructions and Timers on ARMv8-A

Jingquan Ge
Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, China
Department of Computer Science and Engineering, Southern University of Science and Technology, China

Fengwei Zhang∗
Department of Computer Science and Engineering, Southern University of Science and Technology, China
Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, China

ABSTRACT
ARMv8-A processors generally utilize optimization techniques such as multi-layer cache, out-of-order execution and branch prediction to improve performance. These optimization techniques are inevitably threatened by cache-related attacks including Flush+Reload, Flush+Flush, Meltdown, Spectre, and their variants. These attacks can break the isolation boundaries between different processes or even between user and kernel spaces. Researchers proposed many defense schemes to resist these cache-related attacks. However, they either need to modify the hardware architecture, have incomplete coverage, or introduce significant performance overhead.

In this paper, we propose FlushTime, a more secure collaborative framework of cache flush instructions and generic timer on ARMv8-A. Based on the instruction/register trap mechanism of ARMv8-A, FlushTime traps cache flush instructions and generic timer from user space into kernel space, and makes them cooperate with each other in kernel space. When a flush instruction is called, the generic timer resolution will be reduced for several time slices. This collaborative mechanism can greatly mitigate the threat of all flush-based cache-related attacks. Since normal applications rarely need to obtain high resolution timestamps immediately after calling a flush instruction, FlushTime does not affect the normal operation of the system. Security and performance evaluations show that FlushTime can resist all flush-based cache-related attacks while introducing an extremely low performance overhead.

CCS CONCEPTS
• Security and privacy → Side-channel analysis and countermeasures;

KEYWORDS
ARMv8-A; Cache attack; Defense; Flush instruction; Generic timer

1 INTRODUCTION
Nowadays, ARMv8-A devices such as smart phones, tablet, in-vehicle electronic systems and the IoT devices have flooded the market. Moreover, due to higher performance and lower power consumption, many cloud servers [32, 65, 78] based on ARMv8-A have begun to disrupt the data center market. However, like Intel and AMD’s x86 processors, ARMv8-A processors are also suffering a variety of security threats. Among them, cache-related attack is one of the most attractive threats.

∗ Fengwei Zhang is the corresponding author.
We conduct process-to-process Flush+Reload, Flush+Flush, Spectre
which is better than all other software defense solutions.

We implement FlushTime by modifying the Linux kernel, and
perform security and performance evaluations on a Ubuntu system.
We conduct process-to-process Flush+Reload, Flush+Flush, Spectre
attacks, and process-to-kernel Meltdown attacks to evaluate the se-
curity of FlushTime. It shows thatFlushTime has a larger defense
range than other software defense solutions. In addition, we conduct
performance evaluations on instruction calls and APIs respectively.
Moreover, we utilize UnixBench [48] and SPECrate2017 [62] to
evaluate the system performance of FlushTime. Overheads of Flush-
Time are only 1.2% on Unixbench and 0.17% on SPECrate2017,
which is better than all other software defense solutions.

Our main contributions are summarized as follows:

• We propose a scheme to mitigate all flush-based cache-
related attacks by slightly modifying the system kernel.
This scheme does not need to modify the hardware, which is
easy to deploy on existing devices.

• We design a cooperative mechanism between flush instruc-
tions and generic timer. This mechanism not only partially
prevents them from being maliciously exploited by flush-
based cache-related attacks, but also ensures their availabil-
ity in a normal system.

• We conduct security and performance evaluation on the real
hardware platform. The results show that our scheme is not
only more secure than other software solutions, but also
has the lowest performance overhead.

2 BACKGROUND AND MOTIVATION

In this section, we give the detailed descriptions of the cache flush
instructions, high resolution timer/API, and the flush-based cache-
related attacks. Finally, we introduce the existing defense solutions
and their shortcomings in depth.

2.1 Cache Flush Instructions

Most modern processors have cache flush instructions which can
immediately clean a cache line corresponding to a virtual address
of user space. On x86 processors, the instruction cflush [34] can
be utilized to clean up a cache line with a virtual address from user
space. ARMv8-A also has instructions similar to x86 processors.
They are usually called data cache maintenance instructions, such
as DC CIVAC, DC CVAU, and DC CVAC. These flush instructions
can be used in user space (EL0 level), which is the fastest and
easiest way for processors to clean a cache line with a virtual address.
Therefore, these instructions are very useful for the local applications
to guarantee cache coherency. For example, flush instructions are
usually utilized in DMA transfers of local application to ensure
that the data in the cache is consistent with the data in the main
memory. Although the flush instructions provide a fast cache data
cleanup interface, they also bring security threats like flush-based
cache-related attacks.

Fortunately, on ARMv8-A, there is a register SCTLR_EL1 (EL1
System Control Register) [5], which can control the execution privi-
elige level of the cache flush instructions. By default, the bit of
SCTLR_EL1.UCI is set to 1. Under this setting, the cache flush in-
structions can be called at the EL0 level, which are executed in the
user space. In contrast, if the bit SCTLR_EL1.UCI is set to 0, the
cache flush instructions are trapped into EL1 level (kernel space).
Under this setting, when the cache flush instructions are called at the
EL0 level (user space), an interrupt will be generated and the system
will enter the EL1 level (kernel space) to handle this interrupt.

In the Linux kernel, user_cache_maint_handler() [57] is a ready-
made interrupt handler for the cache flush instructions. In our de-
design of FlushTime, the bit SCTLR_EL1.UCI is set to 0, so all flush
instruction executions at EL0 level (user space) are trapped into
user_cache_maint_handler(). We made a small modification to
user_cache_maint_handler() so that the kernel can obtain the re-
duced resolution trigger of current process.

2.2 High Resolution Timers and API

High resolution timer or API is another useful but dangerous sys-
tem resource. On x86 processors, the high resolution timestamp
can be obtained by calling the rdtscl [34] instruction. Correspond-
ingly, ARMv8-A also has several types of high resolution timers,
such as PMCCNTR_EL0 (Performance Monitors Cycle Count Reg-
ister), CNTVCT_EL0 (Counter-Timer Virtual Count Register), and
CNTPTCT_EL0 (Counter-Timer Physical Count Register) [5]. Each
ARMv8-A core has its own PMCCNTR_EL0, CNTVCT_EL0 and
CNTPTCT_EL0. These three timers can be configured as EL0 (user
space) accessible or EL1 (kernel space) accessible. If all the timers
are configured to be accessible at EL0 level, it would bring serious
security threats to the system.

Fortunately, under the default settings of Linux, PMCCNTR_EL0
and CNTPTCT_EL0 can only be accessed at the EL1 level (kernel
space). The CNTVCT_EL0 timer is the only one that is configured
as EL0 accessible by default. Since CNTVCT_EL0 is the only timer
that can be accessed in user space, the clock_gettime() API is also
implemented based on CNTVCT_EL0. The CNTVCT_EL0 timer and
the clock_gettime() API are so easy to use that they are often utilized
by attackers to carry out cache-related attacks.

ARMv8-A has a register CNTKCTL_EL1 (Counter-timer Kernel
Control register) [5], which can control the privilege level that
CNTVCT_EL0 can be accessed. If the bit CNTKCTL_EL1.EL0VCTEN
is set to 0, EL0 level (user space) access to CNTVCT_EL0 will
be trapped to EL1 level (kernel space). Linux kernel has a ready-
made interrupt handler cntvct_read_handler() [86] that handles
CNTVCT_EL0 access. In our design of FlushTime, we set the bit
CNTKCTL_EL1.EL0VCTEN to 0, so reading of CNTVCT_EL0 is
trapped into cntvct_read_handler(). We slightly modified the func-
tion cntvct_read_handler(), which can reduce or restore its own
resolution according to the resolution controller provided by the
kernel.

The function clock_gettime() is a high resolution time API pro-
vided by Linux system by default. The resolution of clock_gettime()
15 years of cache-related attack development, statistical methods. Yarom and Falkner [81] designed the Flush+Reload attack, which ex-

A, the following description will focus on cache-related attacks that without flush. Since FlushTime is a defense scheme based on ARMv8-

At the end of the 20th century, Kocher [41] and Kelsey et al. [36] proposed that cache behavior may pose a security threat. Since then, research on cache-related attacks has developed rapidly. In the first 15 years of cache-related attack development, statistical methods [2, 7, 51, 54, 55, 63, 68, 69, 75] were the most frequently utilized method of cache-related attacks, which are very inefficient because of too much noise. In recent years, to reduce the noise of attacks, the flush instructions are frequently utilized in cache-related attacks [6, 9, 25, 30, 39, 40, 42, 46, 49, 64, 76, 81]. Of course, on some processors, the flush instructions are not available. Researchers have also proposed several types of cache-related attacks [26, 54, 67] that do not require flush instructions.

Below we will introduce cache-related attacks in two categories, namely flush-based cache-related attacks and cache-related attacks without flush. Since FlushTime is a defense scheme based on ARMv8-

Flush-Based Cache-Related Attacks. In 2011, Gullasch [27] pro-

posed a cache-related attack technique, which first utilized clflush instruction on Intel x86 processor. It greatly improves the resolution and reduces the noise of cache-related attack. Three years later, Yarom and Falkner [81] designed the Flush+Reload attack, which extends Gullasch’s technique. The shared LLC (Last-Level-Cache) is the target of Flush+Reload attack, which allows the spy and the victim process to execute in parallel on different execution cores. Subsequently, Flush+Flush attack is proposed by Gruss et al. [25] in 2016. The Flush+Flush attack can be successfully implemented because the x86 clflush execution time of the cached data is higher than the data not cached. Also in 2016, ARMageddon was presented by Lipp et al. [45], which showed that both Flush+Reload and Flush+Flush attacks can be successfully implemented on ARMv8-A.

In addition to cache, branch prediction and out-of-order execution are utilized by most modern processors to improve performance, including ARMv8-A. However, recent research show that these techniques bring huge security risks to the system. Specifically, in 2018, Kocher et al. [40] and Lipp et al. [46] released Meltdown and Spectre attacks to the public, respectively. Subsequently, a series of variant attacks [6, 9, 30, 39, 42, 49, 64, 76] have been released one after another. Among them, there are five variants [6] that pose security threats to ARMv8-A, namely Spectre-PHT [39, 40], Spectre-BTB [40], Spectre-STL [30], Meltdown [46] and Meltdown-GP [6]. These variants are based on Flush+Reload, all of which require flush instructions to successfully perform the attack.

All flush-based cache-related attacks have a common feature, that is, the high resolution time measurement will be executed immediately following a flush instruction. Thus, if the high resolution timer is suddenly unavailable after a flush instruction is executed, the attack would fail. Based on this idea, we design FlushTime, a mechanism for collaboration between flush instructions and generic timers.

**2.3 Cache-Related Attacks**

At the end of the 20th century, Kocher [41] and Kelsey et al. [36] proposed that cache behavior may pose a security threat. Since then, research on cache-related attacks has developed rapidly. In the first 15 years of cache-related attack development, statistical methods [2, 7, 51, 54, 55, 63, 68, 69, 75] were the most frequently utilized method of cache-related attacks, which are very inefficient because of too much noise. In recent years, to reduce the noise of attacks, the flush instructions are frequently utilized in cache-related attacks [6, 9, 25, 30, 39, 40, 42, 46, 49, 64, 76, 81]. Of course, on some processors, the flush instructions are not available. Researchers have also proposed several types of cache-related attacks [26, 54, 67] that do not require flush instructions.

Below we will introduce cache-related attacks in two categories, namely flush-based cache-related attacks and cache-related attacks without flush. Since FlushTime is a defense scheme based on ARMv8-

Flush-Based Cache-Related Attacks. In 2011, Gullasch [27] pro-

posed a cache-related attack technique, which first utilized clflush instruction on Intel x86 processor. It greatly improves the resolution and reduces the noise of cache-related attack. Three years later, Yarom and Falkner [81] designed the Flush+Reload attack, which extends Gullasch’s technique. The shared LLC (Last-Level-Cache) is the target of Flush+Reload attack, which allows the spy and the victim process to execute in parallel on different execution cores. Subsequently, Flush+Flush attack is proposed by Gruss et al. [25] in 2016. The Flush+Flush attack can be successfully implemented because the x86 clflush execution time of the cached data is higher than the data not cached. Also in 2016, ARMageddon was presented by Lipp et al. [45], which showed that both Flush+Reload and Flush+Flush attacks can be successfully implemented on ARMv8-A.

In addition to cache, branch prediction and out-of-order execution are utilized by most modern processors to improve performance, including ARMv8-A. However, recent research show that these techniques bring huge security risks to the system. Specifically, in 2018, Kocher et al. [40] and Lipp et al. [46] released Meltdown and Spectre attacks to the public, respectively. Subsequently, a series of variant attacks [6, 9, 30, 39, 42, 49, 64, 76] have been released one after another. Among them, there are five variants [6] that pose security threats to ARMv8-A, namely Spectre-PHT [39, 40], Spectre-BTB [40], Spectre-STL [30], Meltdown [46] and Meltdown-GP [6]. These variants are based on Flush+Reload, all of which require flush instructions to successfully perform the attack.

All flush-based cache-related attacks have a common feature, that is, the high resolution time measurement will be executed immediately following a flush instruction. Thus, if the high resolution timer is suddenly unavailable after a flush instruction is executed, the attack would fail. Based on this idea, we design FlushTime, a mechanism for collaboration between flush instructions and generic timers.

**2.4 Defenses and Limitations**

**Hardware Defenses.** For Meltdown, Spectre, and their variants, the most effective defense path is to modify the hardware architecture. In the past three years, researchers have proposed many schemes, such as SafeSpec [37], Conditional Speculation [44], SpectreGuard [19], ConTeXT [60], Reuse-trap [16], SpecCFI [43], MuonTrap [4] and GhostMinion [3]. Most of these defenses are effective against speculative cache-related attacks and have a low performance overhead. However, because they require modifications to the hardware architecture, these defenses cannot be deployed on existing devices.

**Software Runtime Defenses.** Before the discovery of Meltdown and Spectre, hardware performance counter was often utilized by re-

searchers [10, 83] to defend against cache-related attacks in real time. However, these runtime defenses are only effective for Prime+Probe and Flush+Reload. They cannot resist the latest Meltdown, Spectre, and their variants. Currently, in the Linux kernel of ARMv8-A, there are three runtime defenses that can effectively defend against speculative cache-related attacks, namely KPTI (kernel page table isolation) [23], Spectre-BTB mitigation [14] and Spectre-STL mitigation [87]. However, Each of these defenses targets only one particular variant. They cannot cover all variants and Flush+Reload attacks. Moreover, these defenses incur a certain performance overhead.

**Static Code Fixing.** Static code analysis/fixing defenses [35, 47, 73] are effective for detecting and defending against flush-based cache-

related attacks. However, fixing the static code brings a significant performance loss to the runtime system [73].

**Browser Defenses.** After Oren et al. [53] successfully implemented their cache-related attack on the browser, browser vendors [11, 82] and W3C [72] have changed the resolution of performance.now from nanoseconds to 5 𝜇s. Unlike browsers, operating system cannot disable the high resolution timer/API because many local applications require it to function properly.

**3 Threat Model**

Our assumptions about the attacker are as follows. First, the attacker can execute her code on the same machine with the victim process.
Attacking does not have the root privilege and cannot use other attack methods to tamper with the kernel code or escalate privileges to obtain sensitive system information. This makes it somewhat difficult for an attacker to obtain the physical address mapping. Although a more sophisticated attacker can still launch cache attacks without obtaining physical address mapping, this requires the attacker to be able to design fully automated methods to generate an eviction set for a given virtual address. In our attack scenario, we assume that the attacker does not have the ability to design an effective eviction strategy. This assumption determines that attackers cannot implement cache-related attacks without flush, such as Prime+Probe, Evict+Reload, MeltdownPrime, and SpectrePrime.

Second, the attacker knows the source code and address layout of the victim process or kernel. There is shared memory between the attacker process and the victim process, so attackers can rely on the flush instructions to clean up the cache lines of the shared pages and leak information. These two assumptions determine that attackers can implement flush-based cache-related attacks, but cannot conduct cache-related attacks without flush.

4 DESIGN OF FLUSHTIME

In this section, we first detail the entire design of FlushTime. Then, we provide details of the relationship between the flush instructions and the generic timer in the FlushTime-enabled system. Finally, we give a description of the two most important parameters in the design of FlushTime.

4.1 Overview of FlushTime

FlushTime is a mechanism for the cooperation between flush instructions and generic timer on ARMv8-A Linux system. According to the introduction of the generic timer in Section 2.2, CNTVCT_EL0 is the only timer that can be directly accessed at the EL0 level (user space). Therefore, in the design of FlushTime, the cooperation between the flush instructions and the generic timer is actually the cooperation between the flush instructions and CNTVCT_EL0. When a flush instruction is called, the resolution of the generic timer will be reduced for a certain period of time. The low resolution duration is granular in context switch().

First, modify the default configuration of Linux to trap flush instructions and generic timer into the EL1 level (kernel space). This is the basis of FlushTime framework. The first reason to trap them is that it is difficult to design cooperation mechanisms for them without trapping them into EL1 level. More secure cooperative APIs can be designed in the library code of user space, but the cooperation mechanism of the flush instructions and generic timer cannot be implemented in user space (EL0 level). Moreover, attackers can execute arbitrary malicious code in the user space of EL0 level, including direct calls a flush instruction or direct access to the generic timer. Thus, they can destroy or bypass the cooperative API mechanism of EL0 level. In the threat model of this paper, the attacker does not have access to the kernel space. Therefore, trapping flush instructions and generic timer into EL1 level and designing a cooperative mechanism in the kernel space is not only easier to implement, but also prevents attackers from destroying or bypassing it.

Second, modify the interrupt handler of the flush instructions. After the flush instructions are trapped into EL1 level, the interrupt will be processed by user_cache_maint_handler(). In the original version of the handler, the flush instructions would be executed directly in EL1 level. In the design of FlushTime, a new operation is added to this handler. This operation is to send the reduced resolution trigger. The trigger signal is used to reset the value of the low resolution delay counter to NumCSLR. We give this low resolution delay counter a name called LRDCounter. The LRDCounter is utilized to count context switch() to ensure that the low resolution state of generic timer lasts for a certain number of context switch(). The main workflow of user_cache_maint_handler() is shown in Algorithm 1.

Algorithm 1: user_cache_maint_handler() in FlushTime

Input: Virtual address to be flushed: vir_addr;
Output: Low resolution delay counter : LRDCounter;
1. Flush the cache line of vir_addr;
2. Store NumCSLR into LRDCounter;
3. Return to EL0;

Third, modify the context switch of the process. When the kernel handles the context switch of the process, it needs to run the context switch() function. In the design of FlushTime, we modify
We give the number of low resolution masked bits a name called \( \text{NumLRMB} \)

at EL1 level and return it to EL0 level. In the design of FlushTime, the cooperative mechanism of the flush instructions and the generic timer is the key to the defense ability. Therefore, in this subsection, we describe in detail the relationship between the flush instructions and the generic timer.

In Linux kernel, the handler `cntvct_read_handler()` handles the interrupt for accessing `CNTVCT_EL0`. In the original version of the interrupt handler, it is only necessary to read the value of `CNTVCT_EL0` at EL1 level and return it to EL0 level. In the design of FlushTime, we modified this handler so that it can automatically adjust the resolution of the returned value. This interrupt handler needs to query the value of `LRDCounter` to decide whether to reduce the resolution of the return value or not. If the value of `LRDCounter` is not 0, the resolution needs to be reduced. Resolution reduction is achieved by masking the least significant bits of the returned value. We give the number of low resolution masked bits a name called `NumLRMB`. Otherwise, the resolution of the value returned to EL0 level is not reduced. Algorithm 2 shows how `cntvct_read_handler()` works.

**Algorithm 2: cntvct_read_handler() in FlushTime**

```
Input:  Low resolution delay counter : LRDCounter;
Output: Time read from CNTVCT_EL0 : CNTVCTime;
1. if (LRDCounter!=0) then
   2. Read CNTVCT_EL0 into CNTVCTime ;
   3. Reduce the resolution of CNTVCTime by NumLRMB bits;
4. end
5. else
   6. Keep the high resolution of CNTVCTime;
7. end
8. Return CNTVCTime to EL0;
```

### 4.2 Relationship between Flush Instructions and Generic Timer

In the design of FlushTime, the cooperative mechanism of the flush instructions and the generic timer is the key to the defense ability. Therefore, in this subsection, we describe in detail the relationship between the flush instructions and the generic timer when FlushTime is enabled. `NumCSLR` is the number of `context_switch()` in low resolution state.

![Figure 2: Relationship between flush instructions and generic timer when FlushTime is enabled.](image)

As shown in Figure 2, in our FlushTime design, after malicious process 1 invokes the flush instructions, the resolution of the generic timer is reduced immediately. This low resolution state will last for `NumCSLR` times of `context_switch()`. If the time interval between two flush instructions is less than `NumCSLR` times of `context_switch()`, the generic timer will remain in a low resolution state and `LRDCounter` will constantly be reset to `NumCSLR`. In other words, the low resolution state of the generic timer will last `NumCSLR` times of `context_switch()` after the last flush instruction is called.

As can be seen from Figure 2, during the time interval between the first and second occurrence of malicious process 1, the flush instruction is called very frequently, and the interval between two flush instructions is much less than `NumCSLR` times of `context_switch()`. Therefore, during this period, the generic timer of all processes is always in a low resolution state. The last flush instruction call occurs at the end of the second occurrence of malicious process 1.

From this point in time, the low resolution state of the generic timer is maintained for `NumCSLR` times of `context_switch()`, and then the high resolution state is restored. On the timeline of the third appearance of the malicious process 1, it is yellow throughout. In other words, during the period of `NumCSLR` times of `context_switch()`, regardless of whether the malicious process 1 calls the flush instructions or not, the resolution of its generic timer is always low. The same is true for other processes such as the normal process k, whose generic timer resolution is always low.

After `NumCSLR` times of `context_switch()` expire, the resolution of generic timer returns to normal. As can be seen from Figure 2, the normal process m is completely green, which means the resolution of its generic timer is always high. This green time period is dangerous because attackers can get high resolution timestamps during this period. To ensure the security of the system, it is especially important to choose an appropriate `NumCSLR`.
4.3 NumLRMB and NumCSLR

In the design of FlushTime, there are two parameters that are critical to security, namely NumLRMB and NumCSLR. The full name of NumLRMB is the number of low resolution masked bits, which represents the number of bits that need to be masked when the generic timer is in the low resolution state. In theory, to ensure the security of FlushTime, the number of masked bits must make the time difference between reading the cached data and the data not cached to be 0. We determined the optimal value of NumLRMB by repeated attack experiments. In Section 6.2, we demonstrate our selection of this parameter in detail with experimental results.

Similar to NumLRMB, the optimal value of NumCSLR is also obtained by a combination of theory and experiment. NumCSLR is shorthand for the number of context_switch() in low resolution state. In a multi-process system, the context switching process of the same processor core will completely clear the data in the cache hierarchy. That is to say, before the context switching process of the same core, there is still sensitive data in the cache. On a single-core processor, context switching for all processes is done on the same core. Therefore, NumCSLR == 1 can ensure the security of sensitive data on a single-core processor. However, on multi-core processors, different processes can execute in parallel on multiple cores. To ensure the security of FlushTime in a multi-core multi-process system, the processor core that invokes a flush instruction must perform more than one context switching process during the low resolution state of the generic timer. Theoretically, to achieve this goal, NumCSLR should be greater than or equal to the number of all processor cores on the platform. The results in Section 6.2 show the correctness of our theoretical analysis of NumCSLR.

5 IMPLEMENTATION

Section 4 focuses on the design principle of FlushTime. In this section, we describe the implementation of FlushTime in five aspects in detail.

Trap flush instructions and CNTVCT_EL0. As described in Section 2.1 and Section 2.2, the bit SCTLR_EL1.UCI controls the trapping of flush instructions, while the trapping of CNTVCT_EL0 is controlled by the bit CNTKCTL_EL1.ELOVCTEN. Under Linux default settings, both flush instructions and access to CNTVCT_EL0 can be executed at EL0 level because SCTLR_EL1.UCI == 1 and CNTKCTL_EL1.ELOVCTEN == 1. In the design of FlushTime, both SCTLR_EL1.UCI and CNTKCTL_EL1.ELOVCTEN need to be set to 0. On the other hand, each CPU core has its own SCTLR_EL1 and CNTKCTL_EL1, so FlushTime needs to modify the settings for each CPU core. To set the two registers for each CPU core, we utilize the ready-made kernel function on_each_cpu() [85], which can call a function on all CPU cores. In addition, in order to make the system more secure, FlushTime should set these two registers as early as possible during Linux boot process. Therefore, in the design of FlushTime, we insert the setting operation of these two registers into the process of Linux multi-core startup. Specifically, FlushTime completes the correct setting of these two registers at the end of smp_init().

Modifications to user_cache_maint_handler(). In the design of FlushTime, in addition to the basic function of cleaning a cache line, user_cache_maint_handler() has to complete one additional operation to trigger the LRDCounter to count. We define a kernel global variable LRDCounter. The meaning of LRDCounter has been described in Section 4.1. We declare this global variable with EXPORT_SYMBOL() and make it visible throughout the kernel. When entering user_cache_maint_handler(), LRDCounter is directly assigned to NumCSLR.

Modifications to context_switch(). The modification of the function context_switch() is very simple, mainly adding the operation of subtracting 1 from LRDCounter. Each time the context_switch() function is executed, LRDCounter is decremented by one. This operation will not stop until LRDCounter reaches 0. Because LRDCounter is a global variable, context_switch() can access it directly after declaring LRDCounter with extern.

Modifications to cntvct_read_handler(). The basic execution flow of cntvct_read_handler() is to read the value of CNTVCT_EL0 and return it to EL0. The value of CNTVCT_EL0 is read by the kernel function arch_timer_read_counter(). In other words, the return value of cntvct_read_handler() is the return value of the kernel function arch_timer_read_counter(). Therefore, we can adjust the resolution of the return value of EL0 by adjusting the resolution of the return value of arch_timer_read_counter(). On the other hand, the interrupt handler cntvct_read_handler() needs to know the value of LRDCounter to determine if the resolution should be reduced. After getting the value of LRDCounter , cntvct_read_handler() will first determine whether LRDCounter == 0 is true. If the result is true, cntvct_read_handler() reduces the resolution of the returned value. Otherwise, cntvct_read_handler() does not adjust the resolution of the returned value. The way to reduce the resolution is to mask the last NumLRMB bits of the return value. In other words, each masked bit is bitwise ANDeD (&) with 0.

Synchronizing the Concurrent Accesses to LRDCounter. In the implementation of FlushTime, it is very critical to solve the concurrent access of LRDCounter. In the initial version of FlushTime, we created a very large two-dimensional array LRDCounter[4096][2]. Each LRDCounter[index][0] stores the pid of the process calling the flush instruction, and LRDCounter[index][1] stores the count of the context_switch of the corresponding process. We set the maximum value of index to 4096 to ensure enough space to store all malicious process pids. This implementation can ensure that each process has its own unique LRDCounter, which can effectively avoid problems caused by concurrent access of multiple processes. However, this implementation has two fatal shortcomings. First of all, this design makes it necessary to traverse all the pids stored in LRDCounter every time the flush instruction is called, which greatly increases the performance overhead of the flush instruction call. Second, even if the maximum value of index is set to 4096, an attacker can use the fork() function to create more than 4096 processes to fill up all LRDCounters, and then create another attack process to launch an effective cache attack. Our approach to solving this problem is to simplify complex problems. Since giving each process a unique LRDCounter is expensive and not secure enough, we assume that all processes calling the flush instruction are the same attacker. In other words, all processes share one LRDCounter. When any process calls the flush instructions, it will reset this unique LRDCounter to NumCSLR. Any process entering context_switch function will decrement this unique LRDCounter by one.
6 EVALUATION

In this section, we show the security and performance evaluation. First, our evaluation environment is introduced in Section 6.1. Then, we explain the selection of optimal values for two important parameters $NumLRMB$ and $NumCSLR$ in 6.2. In Section 6.3, we describe the security evaluation results. Finally, we provide the performance evaluation results in Section 6.4.

6.1 Environment Setup

Our main experimental platform is Huawei TaiShan 200 (Model 2280) server [32]. It has two HiSilicon Kunpeng 920-4826 CPUs [29] with a total of 96 ARMv8-A cores. The platform also has 384GB of DDR4 memory and 2.181TB of hard drive storage. We implemented FlushTime and other defense solutions based on Linux kernel 5.4.128 running in Ubuntu 20.04.2 LTS on TaiShan 200 server. The cross compiler we utilize is aarch64-linux-gnu- with the gcc version of 9.3.0. In addition, in Section 6.2, we also test the two parameters $NumLRMB$ and $NumCSLR$ on another two platforms, Raspberry Pi 4B [18] and ZCU102 [80]. The CPU on Raspberry Pi 4B has four Cortex-A72 cores. The Linux kernel version running on it is 5.15.34, and the compiler running on this board is gcc version of 11.3.0. ZCU102 is designed based on Xilinx Zynq UltraScale+ MPSoC, which has 4 Cortex-A53 cores. The version of Linux kernel running on ZCU102 is 4.14.0-xilinx-v2018.2. The cross compiler we use on this board is petalinux-2018.2.

6.2 Selection of $NumLRMB$ and $NumCSLR$

In the design of FlushTime, the two parameters $NumLRMB$ and $NumCSLR$ play a key role in resisting flush-based cache-related attacks. To maximize the resolution of the generic timer in the dangerous interval, the parameter $NumLRMB$ should be as small as possible while ensuring security. Similarly, $NumCSLR$ should also be as small as possible to ensure that the time interval of the low resolution state is as short as possible. In Section 4.3, we theoretically analyze the optimal values of $NumLRMB$ and $NumCSLR$. In this subsection, we provide experimental results for these two parameters to show that our theoretical analysis is correct.

We utilize the method of repeated attack to select the optimal values of $NumLRMB$ and $NumCSLR$. These experiments are based on two attacks, Flush+Reload and flush-based Specre-BTB attack. In our Flush+Reload attack, the target we selected is the secret key of AES T-Table implementation of OpenSSL 1.1.0. In order to obtain reliable experimental results, we modified the Flush+Reload attack code based on multi-process, so that it has 100 processes during the attack. [1]. Many studies [7, 25, 26, 45, 54] have presented that AES T-Table implementation is very vulnerable to cache-related attacks. We select the $Te0$ table in the AES implementation as the target addresses to crack the secret key. The 256 elements of the $Te0$ table are divided into 16 equal parts. Each 16 elements are in the same cache line. If the correct cache line is found, it indicates that attack succeeds once. According to the random distribution principle of probability, if the success rate of repeated attack is reduced to about 1/16, it means that the Flush+Reload attack fails.

In our Spectre-BTB attack experiments, we port the Kocher’s Spectre-BTB attack code on x86 [40] to our ARMv8-A platform. Like the Flush+Reload attack, the Spectre-BTB attack also executes 100 processes at the same time. Moreover, we slightly modify the code to attack only one byte at a time. Cracking the byte means the attack succeeds once. There are a total of 256 possibilities for a value of a byte. If the success rate of repeated attack is reduced to about 1/256, it indicates that the flush-based Spectre-BTB attack fails.

![Figure 3: The relationship between the parameter $NumLRMB$ and attack success rate on TaiShan 200 server. Both the two attacks are multi-process attacks that execute 100 processes. The red line represents the optimal value of $NumLRMB$.](image)

Figure 3 shows the relationship between the parameter $NumLRMB$ and the success rates of the two flush-based cache-related attacks on TaiShan 200 server. (The parameter $NumCSLR$ is set to 1,000, which is a safe enough value.) As can be seen from Figure 3, the overall trend of the success rates of the two attacks decreases as $NumLRMB$ increases. Starting from $NumLRMB == 6$, the drop in the attack success rate begins to be obvious. This shows that the time difference between reading the cached data and the data not cached decreases significantly after the precision of the timestamp drops by more than 6 bits. When the parameter $NumLRMB$ increases to 12, the success rate of Flush+Reload attack is reduced to about 1/16. Correspondingly, the success rate of flush-based Spectre-BTB attack is reduced to about 1/256 when $NumLRMB == 12$. Therefore, we believe the optimal value of $NumLRMB$ is 12, which is the value we finally select in the design of FlushTime.

![Figure 4: The relationship between the parameter $NumCSLR$ and attack success rate on TaiShan 200 server. Both the two attacks are multi-process attacks that execute 100 processes. The red line represents the optimal value of $NumCSLR$.](image)

The relationship between the parameter $NumCSLR$ and the success rates of the two attacks are shown in Figure 4. Both of the
two attacks are based on multi-process methods, that is, multi-core parallel execution of attack code. As can be seen from Figure 4, when NumCSLR==79, the attack success rate begins to drop sharply. This is because on our 96-core experimental platform, due to the interference of other system processes, the 80th context_switch() has a high probability to occur on the processor core that called the flush instruction. Therefore, sensitive data on the processor core has a high probability of being cleared. As the value of NumCSLR continues to increase, the probability of successful attack continues to decrease. When NumCSLR == 96, the success rate of Flush+Reload attack decreases to about 1/16. At this point, the success rate of flush-based Spectre-BTB attack is reduced to about 1/256. Therefore, the optimal value of NumCSLR that we finally select is 96.

To further study the selection of the two parameters NumLRMB and NumCSLR, we conducted tests on different hardware platforms. Table 1 shows the optimal values of NumLRMB and NumCSLR on different hardware platforms. From the results in Table 1, it can be seen that the optimal value of NumCSLR is consistent with the corresponding number of CPU cores on the platform. However, the relationship between the optimal value of NumLRMB and the hardware platform is not clear. And there is no obvious regularity to follow. According to the estimation of the experimental results, the optimal value of NumLRMB is distributed in the interval of [12, 15]. Analyzing from the hardware level, the main reason affecting the optimal value of NumLRMB is the speed of accessing cache and memory. When the time to access cache and memory is shorter, the number of different bits between cache hit and cache miss is less, so the optimal value of NumLRMB is smaller. The cache and memory access time on TaiShan 200 server platform is better than the other two platforms, so NumLRMB of TaiShan 200 server is smaller than the other two platforms.

### Table 1: Selection of NumLRMB and NumCSLR on different hardware platforms.

<table>
<thead>
<tr>
<th>Platform</th>
<th># ARMv8-A cores</th>
<th>NumLRMB</th>
<th>NumCSLR</th>
</tr>
</thead>
<tbody>
<tr>
<td>TaiShan 200</td>
<td>96</td>
<td>12</td>
<td>96</td>
</tr>
<tr>
<td>Raspberry Pi 4B</td>
<td>4</td>
<td>14</td>
<td>4</td>
</tr>
<tr>
<td>ZCU102</td>
<td>4</td>
<td>15</td>
<td>4</td>
</tr>
</tbody>
</table>

#### 6.3 Security Analysis

To show that FlushTime can effectively defend against all flush-based cache-related attacks, we run comparative attack experiments on Linux systems with different settings. The first system setup is the original Linux kernel, which is the Linux kernel without any defenses configured. The second system setup is the Linux kernel trapping flush instructions and timers. In this setup, both the flush instruction and the generic timer are trapped to EL1, but no cooperative mechanism is implemented. The third system setting is the Linux kernel with FlushTime enabled. To more intuitively show the defense capabilities of FlushTime, we provide the attack results of Flush+Reload, Flush+Flush, and Spectre-BTB in the form of pictures, as shown in Figure 5, Figure 6, and Figure 7, respectively. Finally, we summarize all cache-related attack results in Table 2.

**Flush+Reload and Flush+Flush Attacks.** In our Flush+Reload and Flush+Flush attack experiments, we chose the same target as in Section 6.2, also the secret key of AES T-Table implementation of OpenSSL 1.1.0. In the FlushTime-enabled system, we assign optimal values to NumLRMB and NumCSLR respectively (NumLRMB == 12, NumCSLR == 96). Similar to Section 6.2, we modified the original code of Flush+Reload and Flush+Flush attacks by multi-process to attack with more than 100 processes.

Figure 5 and Figure 6 show the results of the Flush+Reload and Flush+Flush attacks on the Linux system under different settings. The horizontal axis represents the index of the flushed cache line. Meanwhile the vertical axis indicates the offset address relative to the first address of Te0 table in the AES implementation. Since we set the first byte of the secret key k₀ to 0x00, the vertical axis of Figure 5 and Figure 6 also corresponds to (p₀ ⊕ k₀ = p₀). The first byte of plaintext p₀ is increased from 0 to 255 with a step increment of 16. In each offset address of each cache line index, we encrypt the 128-bit plaintext 1,000 times. In Figure 5 and Figure 6, the depth of the color in each grid represents the number of cache hits. If the main diagonal is a straight line of light color, it indicates that our Flush+Reload or Flush+Flush attack is successful.

Our Flush+Reload attack result on the original Linux without any defense is shown in Figure 5a. As can be seen from Figure 5a, Flush+Reload attacks on the original Linux are significantly successful. Figure 5b gives the attack result using the trapped flush instructions and generic timer. These trapped instructions and timer do not have any cooperative mechanism in kernel space. In Figure 5b, the light color of the main diagonal is very obvious, which indicates
that the attack is still successful using these trapped instructions and timer. This result shows that only trapping flush instructions and generic timer into EL1 has no mitigation effect on the Flush+Reload attack. Figure 5c corresponds to the attack result when FlushTime is enabled. From Figure 5c we can see that the color distribution of the grids is very random. There is no obvious color difference between the main diagonal and other grids. These results fully prove that FlushTime is very effective to resist Flush+Reload attack.

Flush+Flush attack experiment is very similar to Flush+Reload. Figure 6 shows the results of Flush+Flush attack on the Linux systems under different settings. Whether there is an obvious color difference between the main diagonal and other grids is the criterion for judging whether Flush+Flush attack is successful. As can be seen from Figure 6a and Figure 6b, the main diagonals in both subplots are lighter in color than the other grids. Therefore, Flush+Flush attack succeeds on the system without any defenses or trapping flush instructions and generic timers into EL1. However, in Figure 6c, all the grid colors are random, and the main diagonal is not an obvious light-colored line. It shows that Flush+Flush attack completely fails when FlushTime is enabled. These comparison results prove that FlushTime is very effective to resist Flush+Flush attack.

**Figure 6:** Flush+Flush attacks on different system setups. It is a multi-process attack that execute 100 processes. The depth of the color corresponds to the number of cache hits in 1,000 AES T-Table encryptions.

**Spectre-BTB Attacks.** In our Spectre-BTB attack experiments, We utilized the same multi-process attack code as in Section 6.2. The target of Spectre-BTB attack is an 11×9 character matrix consisting of ‘F’ and space characters. All the space characters make up the shape of the diamond. This character matrix is sensitive data of the victim process. Figure 7 shows the results of Spectre-BTB attack on different system setups. The character ‘?’ in Figure 7 represents the character that failed to crack. As can be seen from Figure 7a and Figure 7b, all characters in the character matrix become ‘?’ if NumLRMB == 8. And when NumLRMB == 12, all characters in the character matrix become ‘?’. It indicates that FlushTime has a good mitigation effect on Spectre-BTB attacks.

Since the attack results of Spectre-BTB are more intuitive, we tested the impact of FlushTime resolution on the attack success rate in Spectre-BTB attack experiments. Figure 7c and Figure 7d show the results of Spectre-BTB attack on a FlushTime-enabled system. As can be seen from the two pictures, Most characters in the character matrix are ‘?’ if NumCSLR == 96. The resolution of the generic timer is reduced by 12 bits in real time (NumLRMB == 12).

**Figure 7:** Spectre-BTB attacks on different system setups. It is a multi-process attack that execute 100 processes. ‘?’ represents a character that has not been cracked.
a similar impact on Spectre-BTB attacks as NumLRMB. Therefore, we no longer list the comparative experimental results of different NumCSLR here.

<table>
<thead>
<tr>
<th>Attack Classification</th>
<th>Mitigation Capability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prime+Probe [54]</td>
<td>×</td>
</tr>
<tr>
<td>Evict+Reload [26]</td>
<td>×</td>
</tr>
<tr>
<td>Flush+Reload [81]</td>
<td>✓</td>
</tr>
<tr>
<td>Flush+Flush [25]</td>
<td>✓</td>
</tr>
<tr>
<td>Spectre-PHT [39, 40]</td>
<td>✓</td>
</tr>
<tr>
<td>Spectre-BTB [40]</td>
<td>✓</td>
</tr>
<tr>
<td>Spectre-STL [30]</td>
<td>✓</td>
</tr>
<tr>
<td>SpectrePrime [67]</td>
<td>✓</td>
</tr>
<tr>
<td>Meltdown [46]</td>
<td>✓</td>
</tr>
<tr>
<td>Meltdown-GP [6]</td>
<td>✓</td>
</tr>
<tr>
<td>MeltdownPrime [67]</td>
<td>×</td>
</tr>
</tbody>
</table>

**Table 2: Security Evaluation of FlushTime**

**Summary.** In addition to the above three attacks, we have tested all cache-related attacks that can be implemented on ARMv8-A, including cache-related attacks with and without flush instructions. Table 2 summarizes the evaluation results of these attacks. As can be seen from Table 2, FlushTime is powerless to cache-related attacks that do not require flush instructions, such as Prime+Probe, Evict+Reload, SpectrePrime, and MeltdownPrime. However, as described in Section 2.3, Prime+Probe, Evict+Reload, SpectrePrime, and MeltdownPrime attacks require the physical address mapping, which is outside the scope of our threat model. More importantly, FlushTime has good mitigation capabilities against all flush-based cache-related attacks, including Flush+Reload, Flush+Flush, Spectre-PHT, Spectre-BTB, Spectre-STL, Meltdown, and Meltdown-GP, which are the attacks that our threat model focuses on.

### 6.4 Performance Analysis

In this subsection, we first test the comparative latency of calling flush instructions, accessing the generic timer and calling the time API. Then, we use UnixBench [48] to test the performance overhead of critical operations of Linux system. Finally, we provide the performance overhead of real user applications in Linux system using SPEC_CPU 2017 benchmark [62]. In the evaluation experiment of UnixBench and Spec_CPU 2017, we selected three defense schemes, namely KPTI (kernel page table isolation) [23], Spectre-BTB mitigation [14] and Spectre-STL mitigation [87], as the comparison of FlushTime. The reason why these three defense schemes are chosen for comparison is that they have two similarities with FlushTime. First, these three solutions, like FlushTime, do not need to change the hardware, only need to modify the operating system kernel. Second, these three solutions, like FlushTime, can be deployed on ARMv8-A platforms. Based on the above two similarities, this comparative evaluation is valuable.

**Flush Instructions, Generic Timer and Time API.** We first test the latency of FlushTime on calling flush instructions, accessing generic timer and calling clock_gettime() API. Table 3 shows the average time delay for all these operations. We use the instruction ‘MRS X0 CNTVCT_EL0’ to read the value of CNTVCT_EL0. As can be seen from Table 3, the largest increase in time delay is accessing the generic timer CNTVCT_EL0, with an overhead of about 127%. This is mainly because the time delay of accessing the general timer at EL0 is small. After the timer is trapped, the access time is added with the time from EL0 to EL1 and back from EL1 to EL0. Calling clock_gettime() also adds a similar time delay. However, due to the large time delay of calling the time API at EL0, the overhead is only 19%. For the flush instruction, we tested three instructions DC CIVAC, DC CVAU and DC CVAC respectively. Interestingly, after the flush instructions are trapped into EL1, the time delays for calling them did not increase, but decreased. The reason is that calling the flush instructions at EL1 has a shorter latency than calling the flush instructions at EL0. Therefore, despite the increased time to enter and leave EL1, calling these trapped flush instructions is still faster. Overall, the performance impact of FlushTime on calling instructions, calling API, and accessing generic timer is acceptable, and in some respects even better than the original calls and accesses.

**UnixBench.** UnixBench [48] is designed to provide a basic indicator of the performance of a Unix-like system. To test the performance overhead of critical operations of the kernel and the system, we perform UnixBench under 5 system settings. Figure 8 shows the comparative evaluation results of UnixBench. In Figure 8, the five colors represent the five system settings. The white columns represent the original Linux without any defenses. The blue columns are FlushTime-enabled Linux systems. Yellow means that the Linux system only has Spectre-BTB mitigation mechanism [14]. Green means that the Linux system only has the mechanism to defend against Spectre-STL [87]. As can be seen from Figure 8, except for Context Switching, all other individual tests did not increase significantly. The reason for the increase of Context Switching test is mainly because we have increased the count and access of LRDCOUNTER in context_switch() function, which adds extra overhead.

Overall, Spectre-BTB mitigation has the largest total overhead on UnixBench, which is as high as 9.5%. This is mainly because Spectre-BTB mitigation weakens the branch prediction ability of the CPU, resulting in a dramatic increase in Context Switching and Shell Scripts. The total overheads of the Spectre-STL mitigation and the KPTI scheme are 4.9% and 3.5%, respectively. The overhead added by KPTI scheme is mainly concentrated in Shell Scripts, while Spectre-STL mitigation has the greatest impact on Context Switching. Correspondingly, FlushTime’s UnixBench has a total overhead of just 1.2%. This result indicates that FlushTime has much less impact
Figure 8: Evaluation results of UnixBench.

Figure 9: SPEC2017 benchmark results.

on the critical operations of kernel and system than the other three defenses.

**SPEC_CPU 2017.** SPEC_CPU 2017 suites [62] provide a comparative measure of compute-intensive performance using workloads developed from real user applications. Thus, we utilize SPEC_CPU 2017 to compare the impact of FlushTime and other defenses on user application performance. We evaluate all of SPEC_CPU 2017 INT and FP applications under five system settings. These 5 system settings are the same as UnixBench tests. Figure 9 shows the evaluation results of SPECrate2017 benchmark. As shown in Figure 9,
FlushTime has a high overhead in the fotnik3d test. This is mainly because the fotnik3d application calls context_switch() frequently. On the other hand, Spectre-BTB mitigation and Spectre-STL mitigation have more overhead than FlushTime in fotnik3d tests. The reason is that Spectre-BTB mitigation and Spectre-STL weaken the CPU's branch prediction and speculative execution respectively, which greatly affects the performance of fotnik3d.

Overall, in the SPEC_CPU 2017 tests, the Spectre-BTB mitigation has the highest total overhead at 0.67%. In addition to fotnik3d mentioned above, Spectre-BTB mitigation also has a certain impact on the performance of perlbench, gcc, omnetpp, xz and lbm. The reason is that these user applications are sensitive to branch prediction. Followed by KPTI scheme and Spectre-STL mitigation, the total overhead is 0.5% and 0.33% respectively. Among these user applications, lbm and can+ are most affected by KPTI. Correspondingly, Spectre-STL mitigation increases the performance overhead of bwaves and fotnik3d significantly. FlushTime’s total overhead is only 0.17%. This result shows that FlushTime has less performance impact on user applications than the other three defenses.

7 RELATED WORK

In this section, we first present several software runtime defenses against cache-related attacks similar to FlushTime. Then, we detail defense schemes for modifying the hardware architecture.

Software Runtime Defenses. There are several effective studies on software runtime defense against Prime+Probe, Flush+Reload and Flush+Flush attacks. HomeAlone [84] utilizes the same side channel (L2 cache) as attackers to detect Prime+Probe attacks. It allows the tenant to remotely check if the tenant’s own VMs are physically isolated. CloudRadar [83] was proposed for real-time defense against Prime+Probe and Flush+Reload attacks. It combines both anomaly-based and signature-based techniques using hardware performance counters. Secure Collaborative APIs (SCAPI) [20] enables flush and time APIs to cooperate with each other to resist Flush+Reload and Flush+Flush. This cooperation mechanism is similar to that of FlushTime, but FlushTime is more low-level.

There are also some effective software runtime defenses against Spectre, Meltdown, and their variants. Return Trampoline (retoline) [70] are proposed by Google to defend against Spectre-BTB. It is a software mitigation technique that replaces indirect branches with push+return instruction sequences that prevent BTB poisoning. Kernel Page-Table Isolation (KPTI) [23] was proposed to defend against attacks on KASLR [24, 33]. Since it ensures no valid mapping to kernel space in user space, it can also prevent Meltdown. EPTI [31] was proposed to resist Meltdown attack in cloud. It can be applied to unpatched VMs and with less overhead than KPTI. Additionally, Spectre-BTB mitigation [14] and Spectre-STL mitigation [87] are two software runtime schemes in the Linux kernel, which are provided by Linux open source community.

Hardware Defenses. Many researchers modify the hardware architecture to resist Meltdown, Spectre, and their variants. SafeSpec [37] stores side effects of speculative instructions in separate structures until they commit to support leakage-free speculation. Conditional Speculation [44] introduces the concept of security dependence to mark speculative memory instructions which could leak information with potential security risk. SpectreGuard [19] mark sensitive memory blocks using simple OS/library API, and then selectively protect them by hardware from Spectre attacks via low-cost micro-architecture extension. ConTexT [60] requires minimal, fully backward-compatible modifications of applications, compilers, operating systems, and the hardware to offer full protection for secrets in memory and secrets in registers.

Reuse-trap [16] repurposes a classic cache performance metric namely, reuse distance, to capture the activity of an adversary in cache timing channels. It is an efficient cache side channel mitigation framework to record reuse distances during victim accesses and carefully inject noise to mislead the spy from inferring the victim’s activity. SpecCFI [43] utilizes Control-Flow Integrity (CFI) on the committed path, to prevent speculative control-flow from being hijacked to launch the most dangerous variants of the Spectre attacks (Spectre-BTB and Spectre-RSB). MuonTrap [4] prevents the propagation of any state based on speculative execution, by placing the results of speculative cache accesses into a small, fast L0 filter cache, that is non-inclusive, non-exclusive with the rest of the cache hierarchy. It isolates all parts of the system that can’t be quickly cleared on any change in threat domain. GhostMinion [3] is a cache modification built using a variety of new techniques designed to provide Strictness Order, which can comprehensively eliminate transient side channel attacks while allowing complex speculation and data forwarding between speculative instructions.

In addition, there is a category of hardware defense solutions that specifically focus on Cache Set Randomization. ScatterCache [77] makes eviction-based cache attacks impractical because it eliminates fixed cache-set congruences. ClepsydraCache [66] utilizes a novel combination of cache decay and index randomization to mitigate state-of-art cache attacks. MIRAGE [59] is a practical design for a fully associative cache. It is immune to set-conflicts since eviction candidates are selected randomly from among all the lines resident in the cache.

8 DISCUSSION

Other High-Precision Time Resources. Combined with the summary in ARMageddon [45] and other works [22, 58, 61, 79], the high-resolution time resources of ARMv8-A are divided into four categories, namely unprivileged syscall, assembly instruction direct access, POSIX function and dedicated thread timer. The second and third categories are protected by FlushTime. The first and fourth time resources can bypass the protection mechanism of FlushTime. The first category, unprivileged system call, refers to the perf_event_open system call. The time register accessed by this system call is PMCCNT_EL0. Currently, on most ARMv8-A platforms, invoking this system call does not require any privileges. In future work, we can trap access to PMCCNT_EL0 into EL1 kernel space. Then, we can design the coordination mechanism of user_cache_maint_handler() and the handler associated with this system call. This design can achieve the purpose of protecting the system call. The fourth category, dedicated thread timer, is that an attacker can run a thread that increments a global variable in a loop, providing a fair approximation of a cycle counter. This attack method requires the attacker running timing thread on another CPU core in parallel. A possible defense approach is to have the flush instruction to cooperate with the kernel process scheduler. When the flush instruction occurs, the
kernel process scheduler automatically assigns the suspected malicious process’s two threads (attack and timing thread) to the same CPU core. This prevents the timing thread from obtaining the high resolution timing information.

**Cache-Related Attacks without Flush.** Prime+Probe, Evict+Reload, SpectrePrime and MeltdownPrime attacks do not require flush instructions, so the defense mechanism of FlushTime is ineffective against them. Although these attacks require physical address mappings, this restriction does not reduce the threat of these attacks. Because getting physical address mappings in a Linux system is possible [56, 74]. Therefore, in our future work, we expect to continue to enhance the defense capability of FlushTime to make it effective against Prime+Probe, Evict+Reload, SpectrePrime, and MeltdownPrime. We believe that combining the design of HomeAlone [84] and CloudRadar [83] with FlushTime is a feasible solution. Specifically, we expect to add mechanisms to FlushTime that can monitor cache activity. Once the abnormal behavior of the cache is found, the system immediately reduces the resolution of the generic timer. This extended design of FlushTime could defend against Prime+Probe, Evict+Reload, SpectrePrime and MeltdownPrime attacks.

**Deployment in virtualized environment.** There are two types of virtualized environments, one is virtual machines represented by VMware [71], VirtualBox [13], Hyper-V [50] Xen [17] and KVM [12] and the other is virtual containers represented by Docker [15]. If FlushTime is expected to run normally in the above two virtual environments, it is necessary to correctly set SCTLR_EL1 and CNTRCTL_EL1 registers at the hardware level. On this basis, the next FlushTime configuration can be performed. For the first type of virtual machine, the OS kernel running inside the virtual machine is replaceable. Therefore, we only need to replace the old kernel with a new kernel with FlushTime configuration; then FlushTime can run normally in the virtual machine. If we do not have permission to replace the kernel, we can also use kernel hot patch technologies such as kpatch [28] and Ksplice [52] to modify kernel functions user_cache_maint_handler(), context_switch(), and cntvct_read_handler() online. This allows for a successful deployment of FlushTime in a virtual machine. For the second type of virtual container similar to Docker, since the kernel cannot be replaced, kernel functions user_cache_maint_handler(), context_switch(), and cntvct_read_handler() can only be modified online with kernel hot patch technologies such as kpatch and Ksplice. This method allows FlushTime to run normally in the Docker container.

**Impact of Continuous Malicious Flush.** In the setting of FlushTime, a large number of continuous flush instructions will reduce the time resolution of the operating system for a long time, thereby affecting the OS normal function. To investigate the magnitude of this impact in depth, we traversed the source code of the operating system to find all functions that required high-resolution timing. Table 4 shows several representative functions and their file paths. It can be seen from Table 4 that most of the functions that require high-resolution time are used for testing and are not the core functions of the operating system. Therefore, when the time resolution obtained by these functions is not accurate enough, it will not affect the core functions of the operating system. On the other hand, in future work, in order to prevent the impact of continuous resolution reduction on these functions, we can try to design a runtime monitoring module.

The module can monitor these high-resolution test functions in real time, and restore the FlushTime resolution immediately once these functions are found to be called. This real-time monitoring mechanism can effectively prevent FlushTime from affecting the normal functions of the operating system.

**Table 4: Functions that require high-resolution time resources.**

<table>
<thead>
<tr>
<th>File Path</th>
<th>Function Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>tools\testing\selftests\timers\</td>
<td>get_monotonic_and_raw()</td>
</tr>
<tr>
<td>tools\perf\builtin-stat.c</td>
<td>process_interval()</td>
</tr>
<tr>
<td>tools\power\cpu-power-utils\</td>
<td>cpuidle_start()</td>
</tr>
<tr>
<td>tools\testing\selftests\mqueue\</td>
<td>nanosleep_test()</td>
</tr>
<tr>
<td>tools\testing\selftests\mqueue\</td>
<td>perf_test_thread()</td>
</tr>
<tr>
<td>tools\testing\selftests\timers\</td>
<td>test_hrtimer_failure()</td>
</tr>
<tr>
<td>test_sockmap.c</td>
<td>msg_loop()</td>
</tr>
</tbody>
</table>

**Special Timestamp.** As mentioned in Section 6.2, our selected value of NumLRMB is 12. That is, when the generic timer is in the low resolution state, the 12 low-order bits of its timestamp are all 0s. This way of reducing the resolution by setting zero is effective in general. However, some special timestamps invalidate this method of zeroing. For example, when the time interval is [b’nnoon_0000_0000_0nnn, b’nnoon_1111_1111_1nnn], the time interval of low resolution state becomes [b’nnoon_0000_0000_0000, b’nnoon_0000_0000_0000]. As we can see from this example, there is a leak of information in this time interval. To summarize, this happens around the carry of the 13th bit. Although the probability of such a time interval is not high, it still has a certain impact on the success rate of the attack. The fluctuation of the attack success rate after the optimal value in Figure 3 and Figure 4 also confirms the existence of such time information leakage. In our future research work, we expect to deal with the time leakage of these special timestamps without too much impact on the performance of FlushTime.

**9 CONCLUSION**

Flush-based cache-related attacks have become a serious security threat to ARMv8-A-based systems. Existing defense solutions have limited defense coverage, high performance overhead, or cannot be deployed on existing devices. This paper presents FlushTime, a software runtime defense scheme that can be deployed at scale. FlushTime can defend against all types of flush-based cache-related attacks through a novel cooperative mechanism of flush instructions and generic timer. We implemented various types of cache-related attacks on ARMv8-A servers and verified that FlushTime is more secure than other defense schemes. The performance evaluation results show that the overhead of FlushTime is the lowest among compared defense solutions.
ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful comments. This work is partly supported by the National Natural Science Foundation of China under Grant No. 62021015 and Shenzhen Science and Technology Program under Grant No. SGDX202010-3095408029.

REFERENCES

Yevgeniy Sverdlik. 2021. Nvidia Is Designing an Arm Data Center CPU

Raphael Spreitzer and Thomas Plos. 2013. On the Applicability of Time-Driven SPEC.


